1987

Towards a solution to the proper integration of a logic programming system and a large knowledge based management system.

Gorman, John Patrick.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TOWARDS A SOLUTION TO THE PROPER INTEGRATION OF A LOGIC PROGRAMMING SYSTEM AND A LARGE KNOWLEDGE BASED MANAGEMENT SYSTEM

by

John Patrick Gorman

December 1987

Thesis Advisor: C.T. Wu

REPORT DOCUMENTATION PAGE

PORT SECURITY CLASSIFICATION nclassified	1b RESTRICTIVE MARKINGS
CURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
ECLASSIFICATION/DOWNGRADING SCHEDULE	
FORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
AME OF PERFORMING ORGANIZATION l Postgraduate School	6b OFFICE SYMBOL (If applicable) Code 52
	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
DRESS (City, State, and ZIP Code) terey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
ME OF FUNDING/SPONSORING GANIZATION	8b OFFICE SYMBOL (If applicable)
	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
DRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

LE (Include Security Classification)
RDS A SOLUTION TO THE PROPER INTEGRATION OF A LOGIC PROGRAMMING SYSTEM A LARGE KNOWLEDGE BASED MANAGEMENT SYSTEM(u)

RSONAL AUTHOR(S)
ian, John Patrick

PE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month Day)	15 PAGE COUNT
er's Thesis	FROM _____ TO _____	1987 December	48

PPLEMENTARY NOTATION

COSATI CODES		
LD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Expert Systems; Knowledge Based Management Systems, Expert Database Systems; Integration of Expert Database Systems

TRACT (Continue on reverse if necessary and identify by block number)

designing the interface between a database and a logic system with rence such as Prolog, efficiency is the major issue. Presented here are e of the methods that are considered most promising and in which much arch is focused. The first method explores extending an inference ine to include a database manager; the second couples the inference anism with a database management system; and the third extends a data- management mechanism to include inference.

RIBUTION/AVAILABILITY OF ABSTRACT CLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
ME OF RESPONSIBLE INDIVIDUAL C.T. Wu	22b TELEPHONE (Include Area Code) (408) 646-3391 / 22c OFFICE SYMBOL Code 52Wq

M 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Item # 19

Acknowledging up front that no method can be claimed best, the major emphasis of this study will be to determine the strengths and limitations of all three methods and thereby help to clarify many uncertain and sometimes conflicting issues caused by the parallel lines of development from the database and artificial intelligence communities.

Approved for public release; distribution is unlimited.

Towards A Solution To The Proper Integration Of A Logic Programming System And A Large Knowledge Based Management System

by

John P. Gorman
Lieutenant Commander, United States Navy
B.S., Santa Clara University, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1987

ABSTRACT

In designing the interface between a database and a logic system with inference such as Prolog, efficiency is the major issue. Presented here are three of the methods that are considered most promising and in which much research is focused. The first method explores extending an inference machine to include a database manager; the second couples the inference mechanism with a database management system; and the third extends a database management mechanism to include inference.

Acknowledging up front that no method can be claimed *best,* the major emphasis of this study will be to determine the strengths and limitations of all three methods and thereby help to clarify many uncertain and sometimes conflicting issues caused by the parallel lines of development from the database and artificial intelligence communities.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. OVERVIEW

In recent years, great emphasis has been placed on the relationship between logic programming and relational databases. A natural correspondence has been shown between logic programming and expressions of relational algebra and in particular between Prolog facts and tuples of relations [1]. Therefore, great expectations have been raised on the possibility of managing large collections of facts represented in logic programming using some means of retrieving those facts that are stored in secondary storage. An outstanding recent development which promises to make logic progammming an area of major importance is Japan's fifth generation computer system project. The aim of the project is to develop computer systems for the 1990's. This will involve bringing together four currently separate research areas: expert systems, very high level programming languages, distributed computing and very large scale integration technology. It is intended that the very high level languages be based on logic and that the computer achitecture be designed to directly support such languages.

How to best represent the integration of logic programming and relational databases (RDB) is receiving great interest in both the database and artificial intelligence communities because of the widespread applications that can be drawn from such a marriage. However, such a marriage was not conceived in heaven. The intent of this thesis is to examine possible interfaces between an expert system (ES) and database management system(DBMS). The sum of the two is called a knowledge base database

system(KBMS). This chapter will discuss some fundamental differences between logic programming and relational logic/databases. Also, presented will be a discussion of Prolog and why it is viewed as the premier logic programming language. Chapter Two will discuss the issues involved in expanding Prolog to include database facilities. Chapter Three will present an in-depth study of coupling an inference machine with that of a complete database management system (DBMS). Chapter Four presents some issues of expanding DBMS's to include inference and Chapter Five discusses future issues of integrated expert database systems.

B. PROOF THEORY VS MODEL THEORY

Fundamental differences do exist between logic programming and databases in their respective management of data. While some issues can be readily resolved, more fundamental issues are based on theoretical differences such as the model theoretic versus proof theoretic views.

The basis from which relational database and database theory are founded is logic. Such features as integrity constraints and views directly apply to notions of logic. In many instances however, databases are viewed in a model theoretic way. In a model theoretic definition of a database, the state of the database is an interpretation (which must be a model) of the theory. Hence, every evaluation constitutes the computation of a truth value for the query over the current model. During updates, the model changes but not the underlying database schema; consequently, update transactions must preserve database integrity such that the new database will be a model of the theory [1].

Logic programming views databases in light of the proof theoretic view. In the proof theoretic view, facts and deduction rules constitute the theory itself rather than a

model of the more general underlying theory. Queries make up theorems that have to be proven from the theory by using a small number of well founded proof techniques such as resolution. Consequently, update operations change the theory. Time invariance, as discussed above in conjunction with model theoretic, does not exist. Therefore, the theory changes after update operations [2].

The question then is how best to reconcile the two views of databases for the benefit of efficient and intelligent knowledge base management. One way to resolve the differences between the two is through the representation of tables [3]. A logic program is a set of assertions, procedures and a goal. The set of assertions with the same predicate name can be viewed as a table where each row represents a fact. This representation conforms with a relation in relational database. This representation also facilitates applying all assertions at once to an atomic formula and hence the table represents all possible substitution sets for an individual atomic formula [4]. Extending this concept, a conjunction of atomic formulae can also be represented by a table where the individual columns represent the distinct variables/constants in the conjunction and rows represent the set of values the variables can take. Theorem proving, using the resolution principle, can be viewed in terms of table operations producing a new table when one of the atomic formulae in the conjunction is resolved. The advantages of table representation are:

1. Tables can be used to handle sets, where each row corresponds to a substitution set.
2. A relation is often conceived as a table and hence this representation enables a smoother interface with relational database management system (RDBMS).

3. Resolution is in terms of well defined set operations and the possibility of optimization.

4. Negation by failure requires that all solutions be sought before success or failure is declared. Table representation facilitates obtaining all answers and at the same time table sharing reduces memory space needed.

C. INDEXING

Standard relational database systems make extensive use of indices to help search through many facts. A number of techniques, such as B+ trees and indexed sequential access method files, are known and used in the implementation of these sophisticated database systems. The issue of indices and their associated access-methods is necessary to explore in this thesis because of their uses in improving query processing efficiency of integrated logic programming and database systems [5].

Logic programming systems support a limited and fixed amount of indexing. They may implement their own access methods which results in an extension of the logic programming language. This will be discussed in Chapter Two. Secondly, the interface may be coupled with an existing DBMS resulting in two distinct solutions, the "compiled" and "interpreted" approaches. These will be discussed in Chapter Three.

D. PROLOG AS AN IMPLEMENTATION OF LOGIC PROGRAMMING

Prolog interpreter and compilers are the most common logic programming systems available today. In virtually all studies concerning the integration of knowledge programming and DBMS's, Prolog is used as the defacto knowledge programming

language. Prolog has been demonstrated to be easy to learn, write and modify. It is also efficient: interpreted Prolog is about as fast as interpreted Lisp and compiled Prolog is comparable in speed with more conventional languages.

Prolog is a class of implementation of the positive Horn clause subset of logic programming. Prolog programs correspond to hypotheses. Queries correspond to theorems to be proven by the theorem prover using unification. A simple explanation of how Prolog works follows. An example of a Prolog program is:

```
grandparent(x,y):-parent(x,z),parent(z,y).

parent(x,y):- mother(x,y).

parent(x,y):- father(x,y).

ancestor(x,y):- parent(z,y), ancestor(x,z).

ancestor(x,y):- parent(x,y).

father(john,jane).

father(jim,george).

father(chris,john).

mother(april,jane).

mother(heather,april).

mother(mo,john).

mother(april,george).
```

The program consists of a collection of clauses or statements such as ancestor(x,y):-parent(x,y). An atom is a construct such as ancestor(x,y).

Each clause is shorthand for a first order logic formula. The -: is ordinary logical implication and the comma between atoms to the right of the -: stands for logical

conjunction. Clauses with an empty body are called facts. Thus, father (jim,george) states that jim is the father of george. The clauses with non-empty body are conditional and are called rules. A Prolog program is simply a collection of facts and rules.

To compute an answer, a query, a clause with no head, is given to run the program. The answer to a query such as ?grandparent(chris,x) has the property that grandparent(chris,x), with x replaced by the value in the answer, can be deduced from the program. The answer, jane, is deduced using the rule for grandparent, the rule parent(x,y) :- father(x,y) and the facts father(chris,john) and father(john,jane).

During the answering of a query, Prolog is actually proving a theorem. The statement to be proved is the query and the axioms used to prove the statement are the program clauses. Each step in the query answering process uses resolution inference towards proving the query.

Standard Prolog differs from, and complements, pure logic programming in several ways. First, built-in predicates are provided that permit clauses to be read and written to and from terminals and the database, the order of which in the database is not important. For control of the search mechanism, Prolog provides built-in predicates and other features such as cut and fail. And, for support of program development, Prolog provides a few utilities for debugging and tracing programs.

Second, Prolog's proof theoretic resolution is incomplete for the positive Horn clause subset of logic programming. Prolog matching differs in two ways from unification used in resolution. First, Prolog's resolution uses left to right and top to bottom matching. Second, resolution requires that a variable cannot be instantiated to something containing itself. For example, the Prolog expressions

matches(x,x).

equal(example(y,y)).

very well could produce infinite terms and should be avoided.

Prolog evaluation mechanism uses model theoretic as well as proof theoretic views since Prolog does not distinguish facts from predicates and all possible matches are made and printed. Hence, model theoretic answers are obtained simultaneously with proof theoretic answers.

For reasons given above, the relational model is used throughout this thesis when discussing DBMS's. However, many advanced applications find the relational model too sparse, so that models such as the Entity-Relationship model have been developed to capture the semantics of the relationships [6]. It is beyond the scope of this thesis to discuss the issues involved in choosing which model to adopt.

II. EXTENDING PROLOG TO HANDLE SECONDARY STORAGE

A. OVERVIEW

The conceptual difference between a Prolog program and knowledge based program with secondary storage management is the number of facts that reside in secondary storage. This implies that it is impossible to load a deductive database by reading it all at once into main store. Prolog programs are usually small enough to be kept in certain data structures in main store so that they can be easily accessed by the interpreter. In the approach being addressed here, the database must be kept on disk and only those parts of it required to answer the current query are read into main store. Thus, this approach requires file structures similar to a relational database system so that the interpreter can quickly access any fact or rule it requires.

Unlike other approaches in which queries are either handled directly by a relational database or queries are first expressed and controlled by Prolog or some other knowledge based language, this approach senses the close similarities between Prolog and a relational database and seeks to streamline accesses to secondary store by doing away with the relational and implementing all storage in a Prolog framework.

B. CRITICISMS OF PROLOG

1. Lacks Support

The lack of standard database support, such as concurrency, recovery, authorization, and integrity checking, is a criticism of Prolog as a database system, not as a database language.

2. Weak Typing

Prolog does not provide for data definition and typing. The flexibility this provides is desireable in an intensional database where processes are kept in main store. However, there is no mechanism to convey relation schemes, database constraints, or types of attributes. Such information is essential for database design, secondary storage management, optimization and efficient evaluation.

3. Strict Evaluation Strategies

The evaluation strategies used by Prolog are limited. While bottom-up evaluation may be impractical for general clauses, it is quite tractable for many of the clauses encountered in database querying.

4. No Look Ahead

Prolog lacks a mechanism that lets the operating system know of program plans so that the operating system can decide which pages to bump when bringing in new pages.

5. No Secondary Storage Management

By far the biggest problem with using Prolog for databases is its lack of secondary storage management. If Prolog is to be used as a database language, allowances must be made for cases where the database resides in secondary storage because of size and the case where the database system resides at a processor different from the one where Prolog is running.

Concentration on the management of large transfers from secondary storage is a partial solution to to this problem. Consider a simple join statement in Prolog:

```
ans(A,B,C) :- r(A,B), s(B,C).
```

If relations *r* and *s* reside in secondary storage, the programmer must be cognizant of the physical organization of the data. If the relations are not physically clustered, then a block access per tuple is likely. When using Prolog's looping join strategy, a block access could be assumed for each time an *s* -tuple joins with some *r* -tuple, even if *s* is indexed on B.

Common sense strategies would be to read a block of *r* -tuples, to read as many blocks of *s* -tuples as will fit into the remaining memory and to try all joins tuples currently in memory. More blocks of *s* -tuples are brought into main memory until all of *s* is considered. The process is repeated for each block of *r* -tuples. This strategy assumes that most blocks accessed contain many *r* -tuples or many *s* -tuples. Another strategy, based on the sizes of *r* and *s,* is to sort both relations on *B,* if not already sorted, and perform a merge join where only a single pass through *r* and *s* is made.

If tuples are not grouped well or are not retrieved as grouped, alot of data will be moving in and out of core and large virtual memories will not help.

C. ADVANTAGES OF PROLOG

Some of the advantages of Prolog for database programming follow.

1. Easily Augmented

The language can be easily augmented, both in function and in syntax. Special operations, new data structures, and special functions for querying can be readily added by defining new predicates.

2. Straightforward And Flexible

Prolog is a good language for query parsers and translators. It is straightforward to write parsers and tree-transformation programs which makes this language attractive

because of its flexiblity with indices. Programs are easily manipulated as data, making the language a good target for query translation.

3. Simple Virtual Data Definition

Views can be readily added to the database and may reference other views. Views are conceptually the same as relations for querying. There is no need to evaluate views before using them, and computation and retrieval are easily mixed in a view definition.

4. Simplicity

One of the advantages cited of such a system is simplicity when traversing from main to secondary store because of Prolog's good impedence match [2]. For example, in state-of-the-art relational databases, most of the relational ideas are embodied in a powerful, elegant and practical database system. However, most systems, PL/1 and COBOL for example, have host languages. Access to the database is provided by embedded query language statements in a host language such as SQL. SQL however, is so different from PL/1 and COBOL that the resulting combination can be considered unsatisfactory. In addition, programmers will need to know both SQL and one of the procedural system languages. In the approach taken in this chapter, programmers need to know only one language, PROLOG, which can be used to ask and answer queries and also used for running system language programs. Being based on logic, PROLOG is a much more suitable system language for databases than procedural languages such as PL/1 or COBOL. Furthermore, it is possible to give the interpreter sufficient sophistication to optimize queries so that they can be answered with the minimum number of disk accesses.

5. Use Of Rules

This approach also has the advantage of rules. RDBMS's do provide a similar facility, called a view, which is likened to a special case of a rule [3]. Rules share all the advantages that views bring. However, it can be argued that rules are rather more useful and powerful than views. First, rules are at the heart of a deductive database rather than on the outside, as views are in a relational database. Rules can be used in an important way in the data modelling stage of database creation and become an essential part of the data model itself. For example, they provide the flexibility of defining a relation partly by some facts and partly by some rules. Furthermore, rules can be recursive. This is a very useful property which is not shared by views.

In general, relational database systems compensate for their inability to directly express general laws about data by interfacing with conventional programming languages. In other words, a general law, which can be expressed by a single rule, say, in the system being described, will normally need to be expressed procedurally in a relational database system by a program written in a host language. The rule is more explicit, more concise, easier to understand and easier to modify than the corresponding program.

D. CLAUSE INDEXING

In order to efficiently retrieve large amounts of facts in secondary storage a suitable file structure must be found. This is generally called the clause indexing problem.

There are two distinct solutions to this problem. One is the compiled approach and the other is the interpreted approach. Both solutions are examined in detail in another

section of this thesis; suffice it to say that in the compiled approach, the compiler first uses the rules to generate a number of subsidiary queries which can then be answered by looking at the facts. Thus, no fact is accessed until the very last step of the query answering process. In other words, the process is one of pure computation followed by one of pure retrieval. On the other hand, in the interpreted approach, the accessing of facts and the computation are interleaved. Whenever the interpreter needs a fact to continue the computation, a disk access is potentially required (potentially, because the required page may already be in buffer).

E. QUERY OPTIMIZATION

This problem is essentially that of finding an appropriate order, or computation rule, for solving subqueries so that the total cost of answering the query is minimized. Standard PROLOG systems have a fixed left to right computation rule. That is, they always answer the leftmost query first, then the second to the left, etc. In such systems, the query optimization must be done before the query is given to the interpreter. There are essentially two main techniques involved in planning a query.

1. Reorder Clause

An estimate is made of the number of successful matches for each goal in a query, based on the sizes of rules and the number of different values for each attribute. A greedy algorithm is used to reorder the goals by increasing the number of matches. This optimization is similar to that used in SQL, but it is not exhaustive in trying all orderings of clauses, and it doesn't use information on index availablity [7]. This method depends on goals being satisfied by unit ground clauses. The following is an example.

answer(C):- country(C), borders(C,mediterranean),

country(C1),asian(C1),borders(C,C1).

Which is ordered to:

answer(C):- borders(C,mediterranean),country(C),

borders(C,C1), asian(C1), country(C1).

2. Group Independent Clauses

Certain portions of queries may be independent, in that they share no uninstantiated variables at the time of invocation. If a query has two independent parts, there is no point in trying to resatisfy the first part upon backtracking, if the second part fails. Independent portions of queries are grouped with braces and the evaluation mechanism is changed so that portions in braces are skipped over on backtracking. For example, braces can be added to the reordered query above to yield:

answer(C):- borders(C.mediterranean),{country(C)},

{borders(C,C1),{asian(C1)},{country (C1)}}.

Although similar to QUEL in query decomposition, it is not interleaved with evaluation [7].

III. THE COUPLED MECHANISM

A. OVERVIEW

The coupled approach refers to the cooperation of two distinct systems, one for knowledge management (the ES) and one for massive data management (the DBMS). In this approach, the key issue is the interface that allows the two systems to communicate.

Broadly speaking, the key attraction of coupling the ES and DBMS appears to be the production of fast solutions by running two intercommunicating processes: one for the inference machine and one for the facts and statements that reside in secondary store or extensional database (EDB). This sort of coupling could also greatly benefit from the performance advantages of database machines. There are essentially two possibilities when designing this kind of interface: the loosely and tightly coupled.

B. LOOSELY COUPLED

Intuitively, in this approach, the DBMS serves at the request of the ES which processes demands for the DBMS. This approach has also been referred to as the compiled approach [8]. It is based on two distinct phases (eventually repeatedly performed): first a computation on the side of the ES, which, using its knowledge, generates queries for the DBMS; then the execution of the queries on the side of the DBMS and the delivery of the result to the former. Ideally, compiling queries makes use of two processors, one each for the extensional and the other for the intensional so that each machine maintains its own identity. The logic language is essentially devoted to

deductive function and employs theorem proving by using only the intensional axioms without interleaving access to the extensions of the database. The output of the IDB processor is a set of axioms, all of which reference only relations stored explicitly in the extensional database (EDB). After the compiled axioms are produced, deduction is no longer necessary to answer any query on the database and the theorem prover, used to compile the axioms, is no longer necessary. In effect, this decoupling of the EDB and IDB processors relegates the search task over the intensional database (IDB) to the theorem prover, and the inference free computation task over the EDB to the DBMS which uses conventional relational techniques [9].

In order to take advantage of the compiled approach most implementations use some sort of delay tactic in which the query is evaluated either to identify conjuncts of the query belonging to the EDB or to further optimize. Perhaps a link to the DBMS that facilitates the unloading of queries or predicates in optimal form would be required. Automatic loading into a meta-dictionary in order to control access to the IDB from the EDB is also necessary for each specific database. And, some modifications to Prolog are then necessary to permit delay and other optimizations.

A simple example of a use of compilation is as follows. Given a program consisting of a set of rules:

Q1 :- P2, P3.

Q1 :- P2, P3, P4.

Q2 :- P1, P6.

We identify those conjuncts of a query that are directly definable in the EDB with a predefined predicate **ExtDB** so that we have

P1 :- ExtDB(P1).

P2 :- ExtDB(P2).

P3 :- ExtDB(P3).

P4 :- EXTDB(P4).

P6 :- ExtDB(P6).

During the deduction of a query, when an **ExtDB** predicate is to be added to form a new goal, it is placed at the end of the query. Thus, if we have a goal statement of

:- Q1, Q2, Q3.

and a statement with an **ExtDB** pedicate of

Q1 :- ExtDB(Q1),

then the derived goal becomes

:- Q2, Q3, ExtDB(Q1).

So that the entire query is analyzed, all conjuncts that are identified as **ExtDB** conjuncts are moved to the end of the query. In this approach, the fewest changes to Prolog are made. The query is delayed until all possible EDB identifications are made. All EDB conjuncts are passed at once resolving the tuple vs set at a time difference inherent between Prolog and RDBMS.

In modifying the control structure and termination condition, the **ExtDB** predicates are incorporated into the new goal statement being placed at the end of the list of conjuncts in contrast to other predicates which are added to the front of the goal statement. Also, goal statements which start with an **ExtDB** predicate as the first goal in the list of conjuncts must be recognized as a halt condition. When a halt condition arises, the goal statement must be transmitted to the EDB and the RDBMS to obtain the

answers. A meta-dictionary is utilized from which Prolog does a match against the conjuncts and the **ExtDB** predicates. The architecture of this mechanism is given in Figure 1.

Against this approach and the obvious simplicity it provides, we take note of some serious drawbacks. The meta-dictionary would have to be loaded prior to processing for each new EDB . Also, for large EDB's, the meta-dictionary would be proportionately large, requiring constant update in order to sufficiently satisfy all goal queries. This makes it unfeasible. It is also not clear how the concept of views in the IDB side would

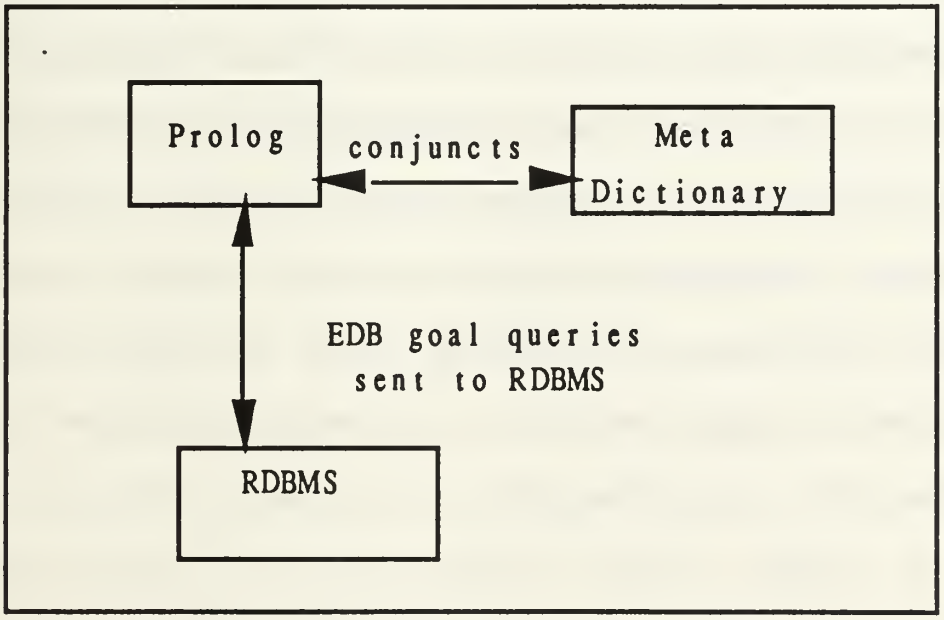


Figure 1. Loosely Coupled Mechanism

be supported. Finally, a problem of consistency may arise if the data collection extracted from the database (which essentially represents a snapshot) is used while the original version is updated.

The attraction of loosely coupled systems is the possibility of optimization in the coupling itself. If query conditions were converted into some normal form, previous to passing these queries to the DBMS, a number of transformation steps could be applied to them to obtain a simplified form of the original expressions. The extra information available at this level and the pattern matching facilities of the logic language provide an optimization mechanism that compliments the one in the DBMS.

The role of integrity constraints as an optimization technique applies nicely to the loosely coupled mechanism. Integrity constraints play no part in the derivation of answers to a knowledge based query in a Horn database. That is, integrity constraints are not necessary to obtain answers to a given existential query in a Horn database [10]. Nevertheless, the role of integrity constraints as semantics (knowledge) expressed over the database has been recognized by Vassiliou and Jarke for its potential optimization possibilities [11]. Integrity constraints can aid/guide and act as heuristics in searching the space for answers. In some cases, they can help terminate queries that have no answers as they violate some database integrity constraints and hence require no search over the database. Integrity constraints can also be used to generate semantically equivalent queries which can be executed more efficiently over the database than the original query. Thus, integrity constraints are valuable to aid the search process or to transform the original query into a set of semantically equivalent queries.

The representation of integrity constraints in clause form is relatively easy. Some examples will suffice to show its simplicity.

"The school that offers knowledge also offers challenges."

:- offer(X,knowledge), offer(X,challenges).

"The course is either in this quarter's schedule or in the

yearly schedule."

spring.schedule(Y), yearly.schedule(Y) :- scheduled(X,Y).

"Only WU teaches computer science courses."

(X = wu) :- teach(X,CS,Z).

Summing the work by Chakravarthy, Fishman and Minker, after compilation of the axioms is complete, a search for predicate matches is made between the axioms and the integrity constraints. If matches are found, the pertinent integrity constraint is added to the matching axiom by using a subsumption algorithm [8]. Ideally, open variables of the axiom are instantiated using the values in the integrity constraint and accesses to the EDB will not be necessary. Partial matches to goal queries are much more likely and, considering very large goal queries, the time saved from calls to the EDB will be substantial. The method is also efficient even in the presence of a large number of integrity constraints and axioms since the integrity constraints are compiled into the axioms only once.

C. TIGHTLY COUPLED METHODS

In the example above, some modification was necessary for proper implementation. One way to avoid modifying the Prolog compiler (interpreter) is to introduce the concept of a meta- language. See Figure 2 for a generic architecture. The

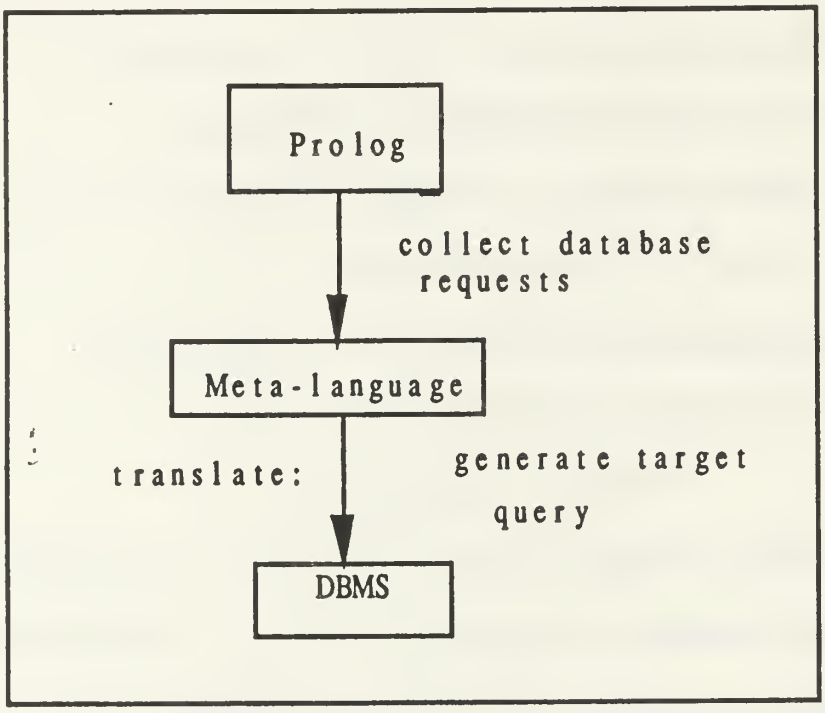


Figure 2. Tightly Coupled Mechanism

sentences in the knowledge language (Prolog) are expressed as terms in the meta-language, along with the new meta-language predicates. Basically, this corresponds to the simulation of the knowledge based language through the high level meta-language constructs. In Figure 2 , the meta-language acts as an efficient translator to form set-oriented queries and optimizes the processing of parameterized views. The flexibility of the meta-language representation arises from the fact that the computation and search rules of the knowledge based language(Prolog) can be modified by suitably defining the meta- language predicates, such as Select, Add, Member, etc. This provides the added advantage of altering the control structure without effecting any

changes to Prolog itself, but only by the redefinition of the meta-language predicates [12].

Perhaps the most obvious advantage of close coupling to the expert system user is the complete transparency of the RDBMS. To illustrate this point consider the relation 'employee' which has two attributes , 'name' and 'salary'. To obtain the salary of Madison, the question is asked:

?- employee(Madison, salary).

regardless of the type of storage used by the relation employee. The fact that employee might be kept as a relation by a RDBMS is completely hidden. This transparency makes close coupling a very attractive choice. In particular, prototype systems could be implemented in the first instance without DBMS facility. Later, when the amount of data exceed certain limits, the DBMS facility could be added. This would not cause changes to the programs on account of the DBMS addition.

In this strategy, the ES consults the DBMS at various points during its operation. In this case an on line communication channel between the ES and the DBMS is required. Queries are generated and transmitted to the DBMS dynamically, and answers can be received and transformed into the internal knowledge representation. Thus, in tight coupling the ES must know when and how to consult the DBMS and must be able to understand the answers.

In a loosely coupled system, the ES has a window on the facts and can access directly only those facts currently loaded on the window. When the data actually held in the window have been processed, the ES asks the DBMS for new data. In the tightly coupled system, the limitations due to the presence of the window are overcome,

and access to data can be performed during the deductive process. The consistency problem identified in loosely coupled systems is avoided since accesses are performed in real time relative to the status of the database.

A naive use of the communication channel would assume the redirection of all ES queries to the DBMS. Any such approach is bound to face at least two major difficulties. The first difficulty is the number of database calls. Since the ES normally operates with one piece of information at a time(record), a large number of calls to a database may be required for each ES goal. Assuming that the coupling is made at the query language level, rather than at an internal DBMS level, such a large number of DBMS calls will result in unacceptable system performance. The number of calls at the query language level could be reduced if these calls result in a collection or set of records. The second difficulty is the complexity of database calls. Database languages usually have limited coverage. For instance, the majority of query languages do not support recursion. For reasons of transportability and simplicity, it may not be desired to include in the coupling mechanism the embedding programming language, say PL/1 or COBOL, a language that would solve the discrepancies in power between the ES and DBMS representations and languages.

The basic scenario for tight coupling a Prolog based ES with an existing relational DBMS is as follows. The user consults the ES with a problem to be solved or a decision to be made; typically this can be expressed as any user friendly language available in ES shells but Prolog predicates will be assumed. Rather than evaluate this user request directly, in a tightly coupled framework the predicate would be massaged into a slightly modified form whose evaluation can be delayed while various

transformations are performed upon it. This process is analogous to a 'pre-processing' stage.

Because of the flexibility the meta-language provides, the design of such a system can take on any aspect. In a general way, such a system would typically be comprised of a translator which would translate the knowledge based language(Prolog) either into a DBMS language(SQL) or one more adaptable towards optimizations. Optimizations would logically be done during this stage. Removal of redundancies to eliminate the execution of unnecessary operations could be implemented using techniques similar to view processing strategies. [13]

Also during this stage, the query could be evaluated to determine which conjuncts would be instantiated by the database in main memory and which to be passed on to the DBMS. Decisions for storage of query results for future reference could be made at this point - particularly important when concerning how to accommodate recursion in Prolog. Finally, another translator must be implemented for translation to and from the data manipulation language if an intermediate language is used.

Close coupling is not without some setbacks. When facts are mapped into relations as implied by the definition of close coupling, operating system resources may be extensively utilized. Processes and pipes could be used to the detriment of the system by slowing the overall response time of the local operating system by consuming a considerable amount of its limited resources. Gains in time, obtained by a very efficient access mechanism, would be almost completely lost in the communication between processes.

Another detriment of close coupling is the overhead required for conversion

between a database source language and Prolog. As an example, consider a database with a relation that stores employees and their managers. Consider a query which, given a single employee, asks who is the highest boss over that employee. This is a standard transitive closure example and, since it cannot be answered by a simple relational calculus query, the need for Prolog becomes obvious. How would a system as described above evaluate this query? The Prolog component would construct a database query to retrieve a single employee tuple, ship it off to the database, which would parse, optimize and execute it and ship the single tuple answer back to the Prolog component. Prolog would then check to see if the employee had a manager, and if so, construct a database query to retrieve a single employee tuple (the manager's), ship it off to the database, etc. The point is that there is a tremendous amount of overhead for what ought to be a simple running of a chain of pointers on disk.

In contrast, loose coupling seems to avoid many of the pitfalls of close coupling. To start with, it requires a minimum effort to implement. As suggested earlier, loose coupling could be implemented by mapping requests in Prolog for service by the DBMS into equivalent expressions in QUEL. Unfortunately, in a loose coupling, regardless of the data manipulation used, no obvious solution exists to the problem of recursion and multiple queries. The problem is rooted in the very large numbers of replies to their respective queries. Depending on the number of communication channels available in the hardware, the problem could be made worse.

1. Interpretive Methods

In the interpretive approach, one interleaves searches of the RDB with deductive steps. The generic approach to this method can be described as having a

Prolog interpreter modified to operate on sets in which a sentence becomes an ordered pair consisting of an ordinary sentence that contains only variables and a set of substitutions. Thus the assertions,

p(a1,b1).

p(a2,b2).

can be represented as a single sentence:

(p(x,y), {[a1,a2]/x,[b1,b2]/y}).

Procedurally speaking, given a goal statement, an atom is selected and a query sent to the RDBMS to retrieve all instances that match the atom. The answer is placed on a stack whereupon Prolog defines it as a statement if the stack is empty or, using deduction, forms a new goal clause if whatever is found on the stack applies and then sends it back to the RDBMS for for further searches. This process is continued until exhausted [3].

a. Set Oriented Approach

This approach can be considered the standard of the interpreted methods. In this approach, the task of generating, including intersection and union of unification sets are relegated to the DBMS which can generate them in an optimized fashion. The knowledge based machine has a set of substitutions instead of a single substitution at each node and the operations are performed on the entire set. This approach fuses together the many branches at the node which arises on account of many unifiers applicable to the same atomic formula. In this approach backtracking is necessary only when multiple procedures are applicable to a single procedure call. The implementation of the set-oriented approach can be conceived in terms of tables at

each node and discussed previously. These tables represent the partial substitution sets applicable at that node.

Chakravarthy and Tran present a technique called table sharing which reduces the total amount of stack space necessary [3]. It is left to the reader to inquire further if interested.

D. SUMMARY

Two modes were distinguished in which a logic programming language can couple with a RDBMS. These are the compiled and the interpretive approaches. In the compiled approach, a theorem prover is used basically to expand an atom in a goal statement so as to generate conjuncts all of which must be looked up in a RDBMS. Two ways are described in which this can be done. The first way required a modification to a Prolog system in how goal atoms are added to a clause. In the second way, the operation is simulated by using meta-level statements directly within Prolog itself.

The interpretive approach solves problems by interleaving searches and deduction. Modifications can be made to a logic language to handle and maintain a set of answers by means of tables.

IV. EXTENDING AN RDBMS

A. OVERVIEW

As previously discussed in Chapter Three, two chief problems of a coupled system are duplication of effort and so called inefficiency stemming from the interface of the expert system front-end and the RDBMS. The question follows then, is there much to gain if the RDBMS is never left? That is, since optimization techniques are already included in a RDBMS, why not extend these systems with new operators to get more expressive power?

The position taken in this chapter is that databases are (or at least can be viewed as) knowledge bases of a certain sort. Databases and knowledge bases try to provide reliable and timely fact management services but with different views. By relating what has already been learned concerning traditional relational databases and logic programs and by considering a database as a very large but limited knowledge base, the use of an extended RDBMS becomes an attractive option presented to us.

B. DATABASE ADVANTAGES

Relational database systems already have many capabilities that mimic logic programming and vice versa. For example, it has already been shown in this thesis that integrity constraints and views can be made to enhance and assume properties of an inference engine. The functions and capabilities in an RDBMS that make this system an attractive choice are listed:

a. The DBMS handles accesses to large databases efficiently.

This has traditionally been the domain of the database system.

b. Multiple users can share the database. Extra concurrency control must be provided in a knowledge base front-end if multiple users are expected.

c. DBMS supports multiple views in a well understood fashion. Prolog supports multiple views of facts through its deduction rules, but their interplay with consistency constraints is not well understood.

d. The structures in a DBMS provide efficient management and programming of large databases. Such structures are entities and relationships and hierarchies.

e. Recovery and security are basic capabilities provided and are mentioned for completeness.

f. Prolog provides no concept of update or transaction consistency. Database transactions can include any number of actions over objects of the database and provide transaction consistency.

C. QUERY EVALUATION

Query evaluation stands at the center of most database power. Database query evaluation sacrifices flexibilty in exchange for increased performance. The proposed solution is to increase RDBMS flexibilty without giving up any performance by allowing the management facility inference capabilities.

Such a system would be a solution to the problem of incompleteness as allowed by

first order logic and consistently seen in Prolog. The problem stems from the ability to state that one of two conditions is true without saying which (using a disjunction operator), or to state that something satisfies a certain condition without saying what that thing is.

Consider, for example, a database of employees, their salaries and department. Queries for the number of employees making over fifty thousand dollars are particularly well suited for traditional database applications but not for Prolog. This brings out an important difference this approach is trying to capitalize on. The Prolog approach to resolve queries leads to resolution by finding solutions through correct representations and indexing. The problem of narrowing in on the data itself has been ignored.

Much research in this area has been centered around the extensions of a database query language such as QUEL to make possible the expression of search algorithms as collections of DBMS commands and to support the selection of an algorithm based on performance considerations.

D. POSTGRES SUPPORT

Most recently, the work by Stonebraker and the implementation of his Postgres system stands out. Postgres supports a collection of alternate algorithms and various implementations of the same algorithm which reside in a library in the DBMS. When a query is introduced, optimization proceeds and a shortest-path algorithm is chosen [14].

Postgres though is an RDBMS that remains within the confines of the model theoretic view. The functions favored by knowledge engineers in a knowledge base, most notably deduction, are still lacking. It is understandable that the proof theoretic view may never be implemented in a database because the proof theoretic approach is not

appropriate to provide for complete data services and for implementing retrieval algorithms over large databases. To be fair, this type of system is designed not to supplant the expert system but is available to be employed by the expert system to provide unified management of data in the knowlege base that resides both intensionally and extensionally.

Postgres represents Stonebraker's implementation of a DBMS that is capable of managing and evaluating rules using either forward or backward chaining. Also, Postgres' rules optimization procedures allow greater flexibility of the RDBMS. Whereas this thesis has been concerned with optimizing predicate matching in a DBMS, now rule management crosses the boundary between the ES and DBMS in either direction. Expert systems using Postgres can perhaps take advantage of this in the compiled stage. This will be discussed further in Chapter Five.

Problem areas still remain in the system. The interfacing query language is even more complex to master than QUEL, making it, as a stand alone system, unattractive to the knowledge engineer who is more comfortable in Prolog or Lisp. Moreover, the accompanying baggage that supports lock implementation, rule mapping, optimization and priorization may cause this system to be slow overall. Stonebraker acknowledges that the rule update system cannot be used to provide view update semantics. So far, only one way mapping from the base relation to the view is provided without requiring an extra semantic definition of what the inverse mapping should be [14]. Work for two way mapping is in progress.

Despite its self-imposed restrictions that prevents full knowledge base utilization, Postgres represents a large stride towards the correct integration of expert systems and

DBMS'. Its portability and anticipated widespread acceptance provide greater flexibility and enhancements for a complete knowledge base.

V. TOWARDS A NEW APPROACH

A. OVERVIEW

In the evolution of the integration of expert systems and the DBMS, it becomes clear that additional functions and new object types must inevitably be represented in either machine to allow optimization of the management of knowledge while maintaining highly deductive powers. In whatever form they take, these new systems should offer a broad range of capabilities without major distruption to the end user. With these goals in mind, these systems should have:

a. Application Independence.

The systems of the future should be general purpose with a spectrum of applicability covering a large set of different application domains.

b. Expandability.

The management of the knowledge, in its different forms as facts, rules and heuristics, must be managed as updatable data and schemas are managed in a modern DBMS today.

c. Alternative Strategies.

The system should have a choice of different search strategies. To help decide which strategy is good for what problem, accessible sets of rules must be able to control search strategy invocation.

d. Multiuser Environment.

Similar to many DBMS', the system should be used for different purposes by different users. To achieve this concept, views must be imported from the database field, with the caveat that knowledge views are much more complex than data views. Multi-user support permits specialists to work both cooperatively, through the knowledge and database, and independently.

e. Efficiency.

This is perhaps the most important factor against which all systems are compared. It seems reasonable to say that the most efficient system to come out will be the one accepted.

In light of the integration aspects already discussed, this chapter explores some further issues related to efficiency and presents its own design for an efficient interface.

The efficient interface between a knowledge base and a conventional database system strives to save the most accesses to secondary storage database. The systems discussed in previous chapters use the paradigm of mapping each pattern matching operation between a knowledge based predicate and the related facts to one query on the relational database. Optimizing queries takes place prior to any interface with secondary storage, e.g., the compiled approach.

Another approach is to save accesses to secondary storage database by never repeating the same database query. This approach requires storing in main memory, in a

compact and efficient way, information about the past interaction with the database elements which have been retreived [15]. Simply speaking, this system would analyze the query and corresponding rules residing in main memory and load those simple relations from secondary storage that might fully or partially fulfill the relations making up the query. For example, suppose the query was posed:

salary(cs,Y):- dept(cs,a,-,C), level(a,Y).

level(Q,A):-dept(Q,a,t,m).

And the database consisted of:

a.dept(cs,a,d,k).

b.dept(cs,b,d,m).

c.dept(ae,a,d,l).

d.dept(cs,a,f,g).

e.dept(oa,a,d,l).

Then a and d would automatically be loaded. Efficiency is gained by partially or fully fulfilling query requirements without going through the overhead of analysis.

B. THE COMBINED APPROACH

On the face of it, this method seems reasonable. However, consider this scenario. An optimizer would have to be employed in the 'pre-analysis' phase to ferret out duplicate predicates and, although the real work of analysis has not yet started, accesses to secondary storage are still taking place. It appears then, that much duplication is taking place in the name of efficiency. Perhaps if the work of the pre-analysis phase were done concurrently with the compilation work in a coupled design then real efficiency would be gained.

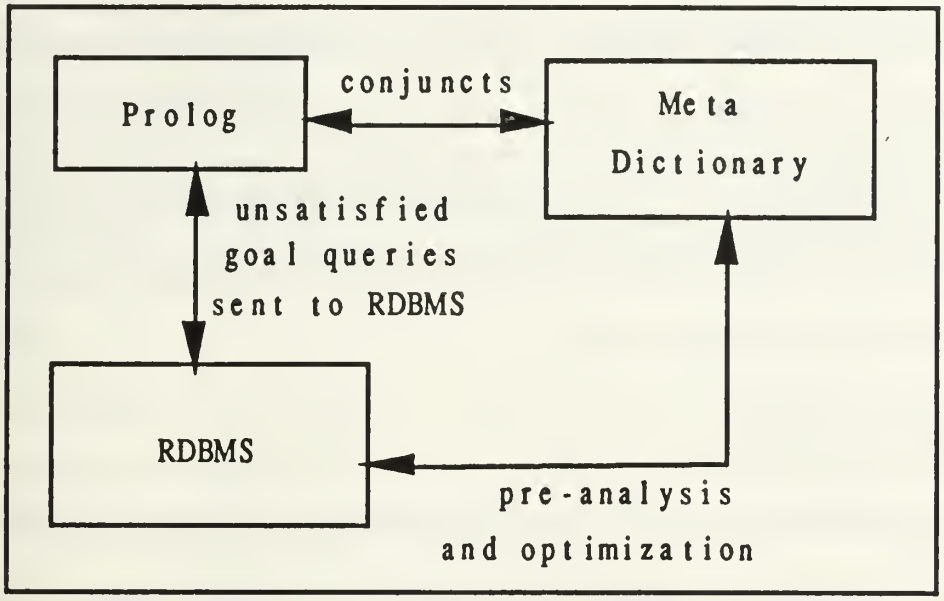


Figure 3. Combined Approach

In Figure three, it can be seen that there is really not much difference between the compiled approach described in Chapter Three of this thesis and what a possible combined compiled/pre-analyzed system could look like. The meta-dictionary, already set up in the coupled approach is crucial in a pre-analysis system for optimization such as keeping track of predicates that have been satisfied by instances in secondary storage. Structures in the meta-dictionary would be necessary to assist in its performance. A shorthand correspondence referring to IDB predicate formulae should be maintained. Implementation of this system could be as a stack that allows the ordered handling of mutliple formulas. Once initiated by the compiled mechanism little upkeep is needed.

Other necessary structures include access-methods to indicate the database access-method for any particular formula and page-pointers for pointing to the current page of the database with respect to the access-method. Access-methods would be either sequential scan or some other technique. Both the access-method device and page-pointers should be maintained at the DBMS level to allow greater flexiblity by the meta-dictionary. The more left at the DBMS level the better overall because of the favorable environment already built and for better portability. The meta-dictionary must still keep track of data-base relations already retrieved.

C. SUMMARY AND SCOPE

This thesis has sketched the concept of integration between a deductive system and a relational database management system illustrating each approach with examples. It was shown that the spectrum of possible mechanisms to link these two components is effectively a continuum from, at one extreme, a single logic-based system that implements both components, to, at the other extreme, two completely separate systems with a strong channel of communication. Which system to employ ultimately is dependent on characteristics attractive to the designer and components already at hand.

Questions continued to be raised by this spectrum of possible mechanisms include: in a coupled system, what is the general architecture for the communication channel between the two components? How can the expert system database calls be translated into the query language of the DBMS? How far can one go in the optimization of queries? And finally, would it be worthwhile to integrate all methods discussed in this thesis into a meta-expert system that combines the expertise of the problem domain with expertise about all connected types.

LIST OF REFERENCES

[1] Gallaire, H., Minker, J., and Nicolas, J. M., "An Overview and Introduction To Logic and Data Bases," in *Logic and Data Bases*, ed. by H. Gallaire (Plenum Press, New York, 1978).

[2] Brodie, M. and Jarke, M., "On Integrating Logic Programming and Databases," in *Proceedings First International Workshop for Expert Database Systems*, ed. by L. Kerschberg (The Benjamin/Cummings Publishing Company, Inc., Menlo Park, 1986).

[3] Chakravarthy, U. S., Minker, J., and Tran, D., "Interfacing Predicate Logic Languages and Relational Databases," Technical Report, TR-1019, University of Maryland, College Park, Maryland, 1981.

[4] Minker, J., "An Experimental Relational Data Base System Based on Logic," in *Logic and Data Bases*, ed. by H. Gallaire (Plenum Press, New York, 1978).

[5] Sciore, E. and Warren, D. S., "Towards An Integrated Database-Prolog System," in *Proceedings First International Workshop for Expert Database Systems*, ed. by L. Kerschberg (The Benjamin/Cummings Publishing Company, Inc., Menlo Park, 1986).

[6] Tsichritzis, D. and Lochovsky, F., in *Data Models*, (Prentice-Hall, Inc., 1982).

[7] Warren, D. H. D., "Efficient Processing of Interactive Relational Data Base Queries Expressed in Logic," *IEEE Seventh International Conference on Very Large Data Bases* 272-281 (1981).

[8] Chakravarthy, U. S., Fishman, D. H., and Minker, J., "Semantic Query Optimization in Expert Systems and Database Systems," in *Proceedings First International Workshop for Expert Database Systems*, ed. by L. Kerschberg (The Benjamin/Cummings Publishing Company, Inc., Menlo Park, 1986).

[9] Lloyd, J. W., "An Introduction To Deductive Database Systems," *The Australian Computer Journal Volume 15, No. 2*, (May 1983).

[10] Reiter, R., "On Closed World Data Bases," in *Logic and Data Bases*, ed. by H. Gallaire (Plenum Press, New York, 1978).

[11] Jarke, M. and Vassiliou, Y., "Coupling Expert Systems with Database Management Systems," in *Artificial Intelligence Applications for Business*, ed. by W. Reitman (Ablix Publishing Corporation, Norwood, 1983).

[12] Missikoff, M. and Wiederhold, G., "Towards A Unified Approach For Expert And Database Systems," in *Proceedings First International Workshop for Expert Database Systems*, ed. by L. Kerschberg (The Benjamin/Cummings Publishing Company, Inc., Menlo Park, 1986).

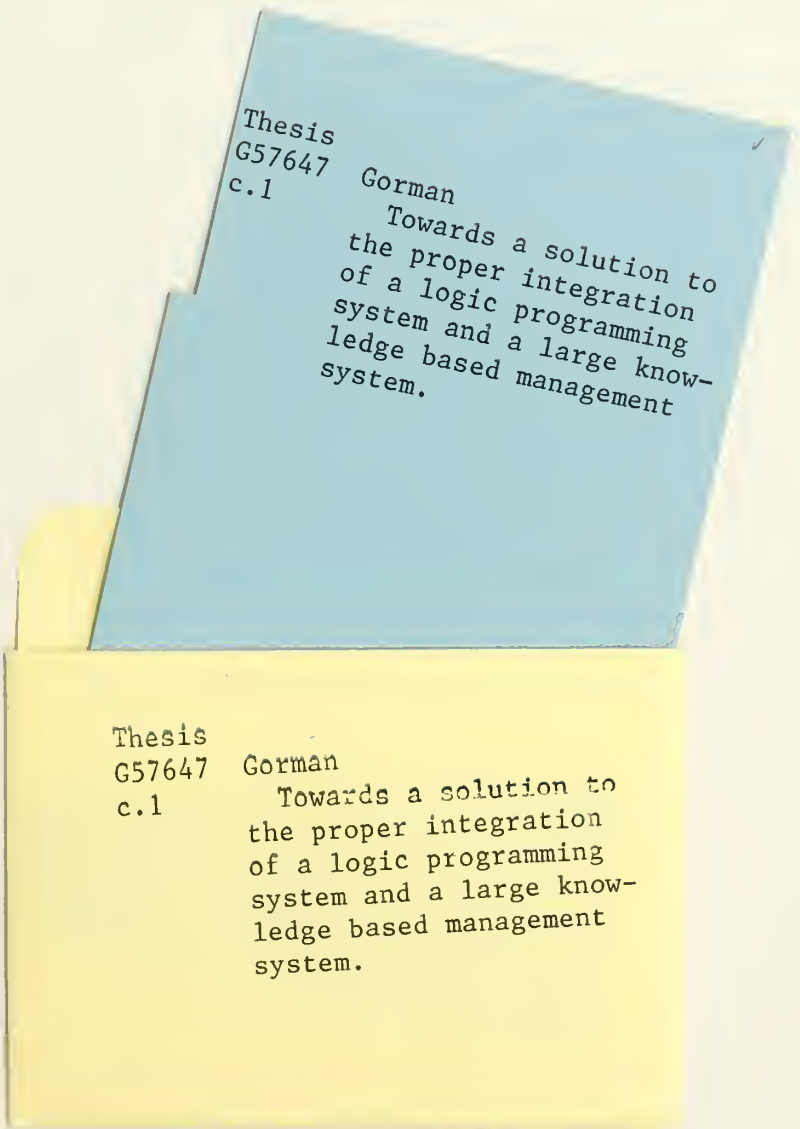
[13] Rosenthal, A. and Reiner, D., "Querying Relational Views of Networks," *Proceeding IEEE COMPSAC* (1982).

[14] Stonebraker,ed., Michael and Rowe,ed., Lawrence A., "The Postgres Papers," Memorandum No. UCB/ERL M86/85, University of California, Berkeley, Berkeley, June 1987.

[15] Ceri, S., Gottlob, G., and Wiederhold, G., "Interfacing Relational Databases and Prolog Efficiently," in *Expert Database Systems, Proceedings of the First International Conference on Expert Database Systems*, ed. by L. Kerschberg (Insitute of Information Management, Technology and Policy, University of South Carolina, Charleston, 1986), p. 141-153.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
5.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor C. T. WU, Code 52Wq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2



Thesis
G57647
c.1
Gorman
Towards a solution to the proper integration of a logic programming system and a large knowledge based management system.